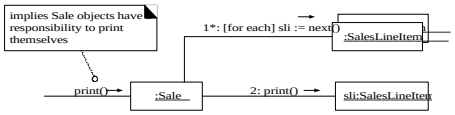
GRASP Patterns (Ch. 18)

- Responsibility:
  - "A contract or obligation of a type or class"
- Two type of responsibilities
  - knowing
    - about private encapsulated data
    - things that can derive or calculate
  - doing
    - something itself
    - initiating action in other objects

1

Responsibilities & Interactions

- Methods Implement Responsibilities



2

Patterns

- Named "problem/solution" pairs
- Primitive examples from OOP:
  - Model - View - Controller (MVC)
  - Object - Attribute - Value (OAV)
- GRASP (Larman)
- Gang of Four (GoF; Gamma, et. al.)

3

Basic GRASP Patterns

- Expert
- Creator
- Controller
- High Cohesion
- Low Coupling
- Learn the other patterns after these are mastered

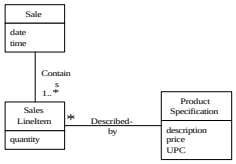
4

Expert Pattern

- The most commonly used pattern
- Assign a responsibility to the information expert
  - the class that has the information necessary to fulfill the responsibility
- Example:
  - In POST, who should know the grand total of a sale?

5

Grand Total Expert?

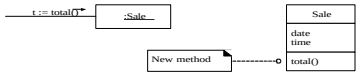


- Who knows about *SalesLineItem*s and their totals?

Answer: *Sale*

6

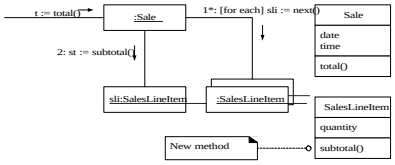
What Else is Needed?



- What constitutes the line item subtotal?
  - `SalesLineItem.quantity`
  - `ProductSpecification.price`
- Who is the expert on both of these?
  - Answer: *SalesLineItem*
- *Sale* needs to see subtotal msgs from *SalesLineItem*

7

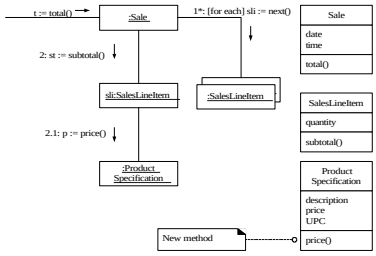
Resulting Collaboration Diagram



- *Sale* needs to send subtotal msgs to each *SalesLineItem* and sum the results
- In turn, *SalesLineItem* must send a *price* msg to *ProductSpecification*

8

Resulting Design



9

Expert - Conclusions

- In the real world, items don't tell you their price; line items don't tell you their total
  - But in O-O world, they do!
  - This principle is called "Animation" or the "Do it Myself" principle
- Also works in the workplace (real world)
  - Who puts together the profit/loss statement?
  - Ans: the person in accounting with all the data

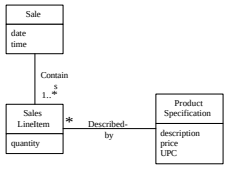
10

Creator Pattern

- Assign class B the responsibility to create an instance of class A if one of the following is true:
  - B aggregates A-type objects (collects)
  - B contains A-type objects (owns)
  - B records instances of A-type objects
  - B closely uses A-type objects
  - B is an expert w/r/t creating A-type objects
    - B has the initializing data that will be passed to A when it is created
- In all cases, B is the creator of A-type objects!

11

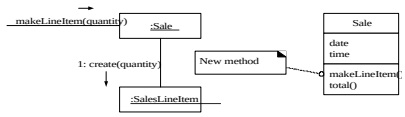
Who Should Create SalesLineItem Instances?



- Answer: *Sale* - since it aggregates many *SalesLineItem* instances
- Requires a *makeLineItem* method be defined in *Sale*

12

Resulting Design



- Question: Who would best be able to create a Payment object?
  - Answer: *Sale* - since it aggregates necessary data (like total sale) to generate the object.

13

Low Coupling Pattern

- Formalized by Constantine & Yourdon
  - *Structured Design, 1974*
- *Coupling* is the degree to which modules (separate functions or methods) are inter-related.
- Low coupling is desired
- Levels of coupling are identified

14

Levels of Coupling / Cohesion

Best

- Data - Low
- Stamp
- Control
- Common
- Content - High

- Informational - High
- Functional
- Sequential
- Communication
- Temporal
- Logical
- Coincidental - Low

Worst

15

Data Coupling

- The lowest level of coupling (the best form)
- Also, the most common form of coupling
- Passing parameters between modules as data
- The fewer parameters passed, the lower the coupling
- Return values are used as data in the caller

16

Stamp Coupling

- Selected global variables are shared by multiple modules or routines
- If unexpected values arise in these globals, who is responsible?!
  - Requires a “watch” feature of a symbolic debugger to get a handle on who’s responsible
- Use of global variables is discouraged when parameter passing would suffice

17

Control Coupling

- Involves passing of control flags (either through parameters or global variables) as data between modules.
- The return value of the function determines the flow of control in the calling module
  - `if (func()) then … else …`
- What if there’s an error in `func()`?!

18

Common Coupling

- Modules are dependent on one anther (bound together) by global data blocks
- Fortran common blocks are the reason for this level of coupling
  - Common blocks are blocks of shared memory, with every holder of the common stmt having read/write access to anything in the block
  - Large amounts of data are typically shared, often to usurp parameter passing of arrays.

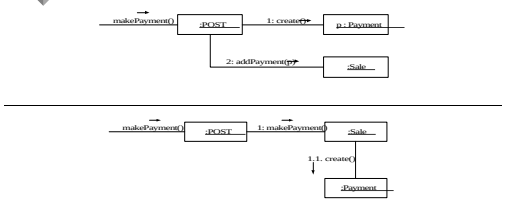
19

Content Coupling

- The highest form of coupling (worst form)
- Occurs when one module modifies code and/or data in another module
- Some re-enterant code from assembly lang. exhibit this form of coupling
- Can be nearly impossible to detect and debug (because it disappears!)

20

Consider Two Designs



- Which has lower coupling?
  - Answer: The 2nd design

21

High Cohesion Pattern

- Formalized by Constantine & Yourdon
  - *Structured Design, 1974*
- Cohesion is the degree to which statements within a module (functions or methods) are inter-related
- The higher the cohesion, the better
- Levels of cohesion are identified

22

Levels of Coupling / Cohesion

Best

- Data - Low
- Stamp
- Control
- Common
- Content - High

- Informational - High
- Functional
- Sequential
- Communication
- Temporal
- Logical
- Coincidental - Low

Worst

23

Informational Cohesion

- A concrete realization of Data Abstraction
  - classes in C++/Java/other OO langs.
- The module contains data structure(s) and algorithm(s) which are implemented to provide one concept
  - Also known as Abstract Data Types, Packages (Ada), Envelopes (Pascal Plus), Clusters (CLU)
- Everything works together to provide the abstraction

24

Functional Cohesion

- Elements within a module are related in performing some mathematical-type function
  - `sin(x)`, `sqrt(x)`, `pow(x, y)`, etc.
- Functional dependence within a module is a common, sought-after feature

25

Sequential Cohesion

- Elements are grouped together (in the design of the system) based on data flow
  - Data flow is considered a stronger design issue than control flow!
- Occurs when the output from one element is input to the next element
  - Examples:
    - Read next transaction, update master file
    - "pipes" in Unix, network "sockets"

26

Communication Cohesion

- Elements refer to the same set of input or output data at the same time in processing
- Stack operations exhibit this form of cohesion
  - `push()`, `pop()`, `top()`, `new()`, `empty()`
- Data access is also coordinated in time

27

Temporal Cohesion

- Group elements together that can all be performed at the same time
- Initializing variables, initializing the state of an object
- Temporally cohesive elements tend to also be logically cohesive

28

Logical Cohesion

- Elements are grouped together because they perform the same logical type of operation
- Intializing all variables
  - also temporally cohesive
- Perform all I/O operations
- Process all records from a file
- Display error messages

29

Coincidental Cohesion

- The lowest form of cohesion (worst)
- Elements within a module have no apparent relationship!
- Sometimes results when one is forced to upgrade a program to a new language
  - "Modularize" an assembly program Into C
  - Convert a functional program to an O-O language (like Java, which only allows member function, no global functions)

30

Coupling & Cohesion

- Are inversely related
  - as one increases, the other decreases
  - Low coupling, high cohesion is good
  - High coupling, low cohesion is bad
- Low cohesion means the module should delegate its work to subclasses
  - Just like the person who takes on too much in life, things become complicated & nothing is done well!

31

Controller Expert

- Assign responsibility of handling system events to a class representing:
  - the overall system (*Facade controller*)
  - the overall business or organization (*Facade controller*)
  - something in the real world that is actively involved in the task (like a person playing a role) (*Role controller*)
  - an artificial handler of all system events of a use case, usually named "<UseCaseName>Handler" (*Use Case controller*)
- System events are generated by external actors

32

Controller Objects

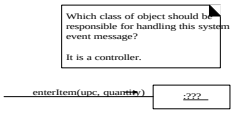
- A *controller* is a non-user interface object responsible for handling a system event.
- A *controller* defines the method for the *System* object
- POST system events & their operations:

System
endSale() enterItem() makePayment()

System operations do not have to be fulfilled in the System class. During design, controller classes are often delegated the responsibility for system operations

33

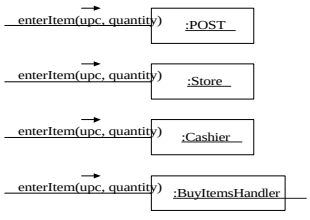
Who Should be the Controller?



- POST
  - represents the overall "system"
- Store
  - represents the business organization
- Cashier
  - represents actor that plays a role in the task
- BuyItemsHandler
  - represents the handler of the system operation in the use case

34

Controller Choices



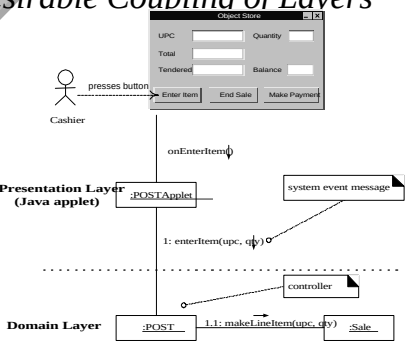
35

Answer: It Depends!

- The same controller should be used for the system events of one use case
  - For instance, *makePayment()* comes after *endSale()*
- Facade controllers are suitable when there are only a few system events.
- Only the role controller should be avoided
  - Bloated controllers; poor Design
- Presentation layer does not handle system events

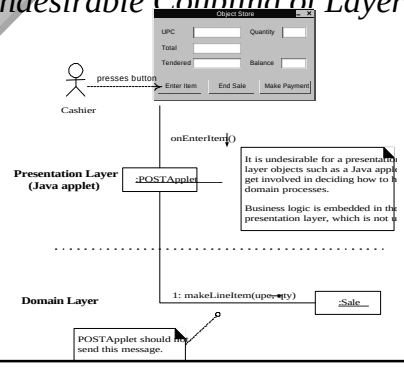
36

Desirable Coupling of Layers



37

Undesirable Coupling of Layers



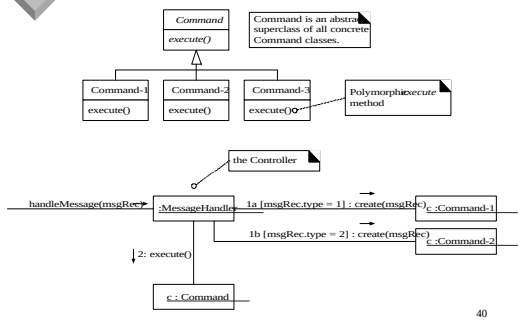
38

Command Pattern

- Useful when msgs are received, not through a UI, but from a msg handling system
  - Telecommunications switching systems
  - CORBA (object DB) architectures
  - other protocol handlers (mail, ftp, etc.)
- Each command (msg) has its own class
  - each with an *execute()* msg
  - polymorphism allows different subclasses to implement the execution

39

Command Controller Pattern



40

Related Patterns

- Command - In a msg handling system, each msg may be represented and handled by a separate Command object
- Facade - Choosing an object representing the entire system or business organization
- Forwarder-Receiver - A Siemens pattern useful for message handling systems
- Layers - Another Siemens pattern. No domain logic in the presentation layer is part of this
- Pure Fabrication - A GRASP pattern. A use case controller is a kind of Pure Fabrication

41

CRC Cards *(Beck & Cunningham, 1989)*

- Class, Responsibility, Collaboration Cards
- Each card has a Class name & list of responsibilities
- Each person holds onto cards and "plays the role" of that class in collaborating to perform a task or implement a use case.
- Collaboration and Class diagrams capture the dynamics that occur in the role playing

42